



Carnegie Mellon  
Software Engineering Institute

# Model-Based Verification: Guidelines for Generating Expected Properties

David P. Gluch  
Santiago Comella-Dorda  
John Hudak  
Grace Lewis  
Chuck Weinstock

**DISTRIBUTION STATEMENT A:**  
Approved for Public Release -  
Distribution Unlimited

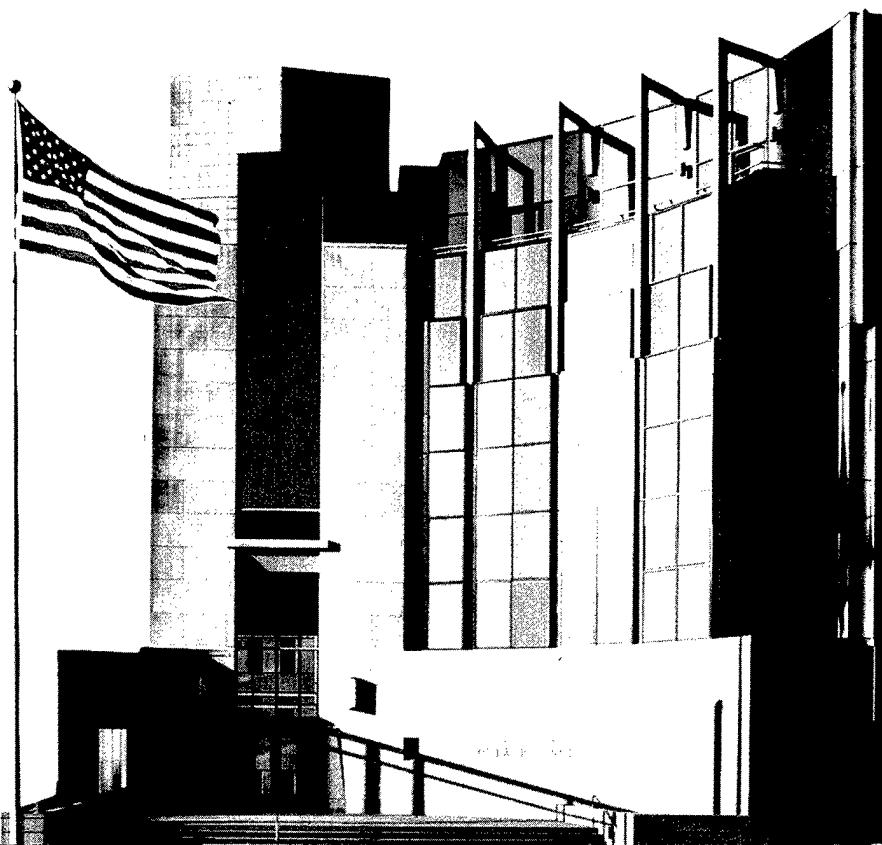
*January 2002*

## Performance Critical Systems

Unlimited distribution subject to the copyright.

**Technical Note**  
CMU/SEI-2002-TN-003

20020221 032



Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of "Don't ask, don't tell, don't pursue" excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2002 by Carnegie Mellon University.

**NO WARRANTY**

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (<http://www.sei.cmu.edu/publications/pubweb.html>).

---

## **Contents**

<b>Abstract.....</b>	<b>v</b>
<b>1    Introduction .....</b>	<b>1</b>
<b>2    Characterizing Expected Properties .....</b>	<b>3</b>
2.1 Implicit vs. Explicit Statements.....	4
2.2 Application Domain Considerations .....	5
2.3 Generic vs. Detailed Statements.....	5
2.4 Whole vs. Part Statements .....	6
<b>3    Generating Expected Properties .....</b>	<b>7</b>
3.1 Focusing the Generation Process.....	7
3.2 Partitioning of Expected Properties.....	7
3.3 Capturing Expected Properties .....	8
<b>4    Expected Properties and Claims .....</b>	<b>10</b>
4.1 The Relationships between Expected Properties and Claims .....	11
4.2 Classifying Expected Properties and Claims.....	12
<b>5    Conclusions.....</b>	<b>15</b>
<b>References .....</b>	<b>17</b>



---

## **List of Figures**

Figure 1: Model-Based Verification Process and Artifacts .....	1
Figure 2: Model Checking .....	3
Figure 3: Requirements and Expected Properties Elicitation Processes .....	8
Figure 4: Expected Property Transformations .....	11



---

## **Abstract**

This report presents a basic set of guidelines to facilitate the generation of expected properties in the context of Model-Based Verification. Expected properties are natural language statements that express characteristics of the behavior of a system—characteristics that are consistent with user expectations. Through model checking, expected properties of a system, formally expressed as claims, are analyzed against the model. This analysis can detect inconsistencies between models of the system and their expected properties and identify potential system defects.



# 1 Introduction

Model-Based Verification (MBV) is a systematic approach to finding defects (errors) in software requirements, designs, or code [Gluch 98]. The approach judiciously incorporates mathematical formalism, in the form of models, to provide a disciplined and logical analysis practice, rather than a “proof of correctness” strategy. MBV involves creating essential models of system behavior and analyzing these models against formal representations of expected properties.

The artifacts and the key processes used in Model-Based Verification are shown in Figure 1. Model building and analysis are the core parts of Model-Based Verification practices. These two activities are performed using an iterative and incremental approach, where a small amount of modeling is followed by a small amount of analysis. A parallel compile activity gathers detailed information on errors and potential corrective actions.

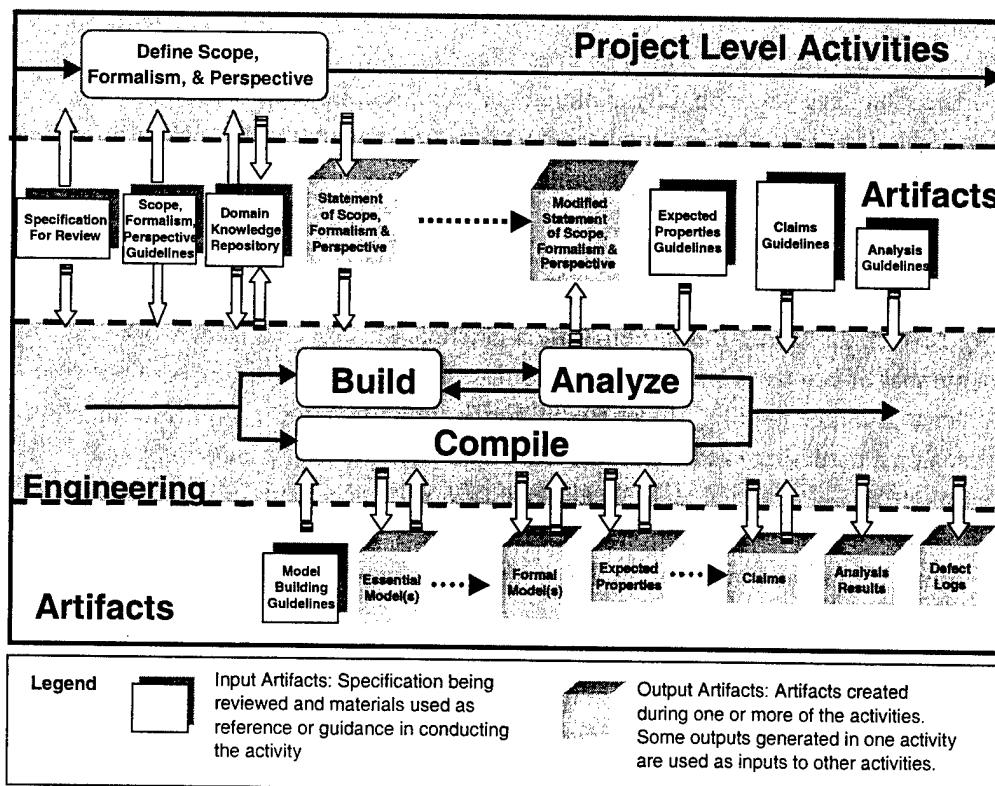


Figure 1: Model-Based Verification Process and Artifacts

An essential model is a simplified formal representation that captures the essence of a system, rather than provide an exhaustive, detailed description of it. Through the selection of only critical (important or risky) parts of the system and appropriately abstracted perspectives, a reviewer, using model-based techniques, can focus the analysis on the critical and technically difficult aspects of the system. Driven by the discipline and rigor required in the creation of a formal model, simply building the model, in and of itself, uncovers errors.

Once the formal model is built, it can be analyzed (checked) using automated model-checking tools. Within this analysis, the user identifies potential defects both while formulating claims about the system's expected behaviors and while formally analyzing the model using automated model-checking tools. Model checking has been shown to uncover the especially difficult-to-identify errors: the kind of errors that result due to the complexity associated with multiple interacting and interdependent components [Clarke 96]. These include embedded as well as highly distributed applications.

A variety of different formal modeling and analysis techniques are employed within Model-Based Verification ([Gluch 98, Clarke 96]). The choices are based upon the type of system being analyzed and the technological foundation of the critical aspects of that system. This decision on the technique(s) involves an engineering trade-off among the technical perspective, formalism, level of abstraction, and scope of the modeling effort.

The specific techniques and engineering practices of applying Model-Based Verification to software verification have yet to be fully explored and documented. A number of barriers to the adoption of Model-Based Verification have been identified including the lack of good tool support, expertise in organizations, good training materials, and process support for formal modeling and analysis.

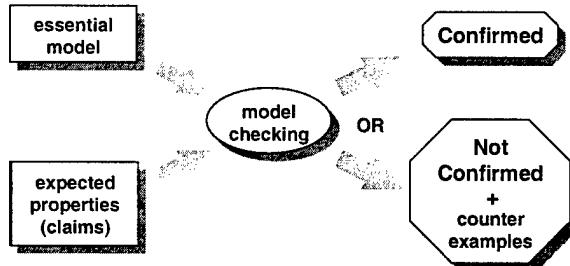
In order to address some of these issues, the Software Engineering Institute (SEI) has created a process framework for Model-Based Verification practice. This process framework identifies a number of key tasks and artifacts. Additionally, the SEI is working on a series of technical notes that can be used by Model-Based Verification practitioners. Each technical note is focused on a particular Model-Based Verification task, providing guidelines and techniques for one aspect of the Model-Based Verification practice. Currently, the technical notes that are planned address abstraction in building models, generating expected properties, generating formal claims, and interpreting the results of analysis.

This technical note addresses the search and definition processes (generation process) for expected properties. Expected properties are natural language statements about the characteristics of a system's behavior—characteristics that are consistent with user expectations. The generation activity involves the participation of domain experts as well as software engineers and requires a systematic approach not unlike requirements elicitation techniques. Once generated, expected properties are expressed as formal claims. These formal claims are compared against a model of the system using model-checking tools.

---

## 2 Characterizing Expected Properties

Building essential models and analyzing models against expected properties are fundamental practices in Model-Based Verification. The automated analysis process of model checking is shown in Figure 2. The results of the analysis are a confirmation of the correctness of the properties or a statement that the properties are not true for the model. In many cases when a property is not true, a counter example showing a violation of the property is included in the output. Model-checking tools (e.g., the Symbolic Model Verifier [Clarke 95, McMillan 92]) leverage the formal aspects of models to automatically analyze them, determining whether or not specific system properties, expected properties, are valid for the model.



*Figure 2: Model Checking*

Expected properties are natural language statements about the behavior of a system. In model checking, models can be thought of as formal representations of behavior or structure<sup>1</sup> and expected properties as the characteristics of that behavior and structure. Expected properties are complementary to models of a system and often describe properties that reflect the requirements but are not explicit in a model.

As an example of the distinction between models and expected properties, consider a system that involves the sharing of resources among different functional components. A requirement may be that only one of these components at a time should access a given resource. A design approach may be to use an algorithm that defines how processes should coordinate their access to a shared resource. In a Model-Based Verification analysis of the design, a model that reflects this algorithm may be built. The model portrays the behavior that is dictated by the algorithm in some formal modeling representation. But nowhere in the model is there an explicit statement that (1) only one process should have access to a shared resource at a time

---

<sup>1</sup> We have focused on behavioral models. Models that portray the structure of a system can be checked for the integrity of their structure as well as for the behavioral implications of the model (e.g., Alloy [Jackson 00] and NitPick/Ladybug [Jackson 96a, 96b]).

or that (2) if a process requests access, it eventually will get access to the shared resource. Nevertheless, mutual exclusion and fairness would be expected properties of such a system. Through model checking the model should demonstrate them in its portrayal of the system's behavior.

Characterizing expected properties is not simple. Expected properties complement models in that expected properties portray system behavior in a different, often more general or abstract way. The next subsections discuss the special nature of expected properties and how they complement models. This characterization will help identify expected properties for a given essential model.

## 2.1 Implicit vs. Explicit Statements

The implications of an explicitly stated requirement are good candidates to become expected properties. These implied attributes of a system are often not recorded in any document. Some of these attributes involve general engineering common sense (the system should not crash every five minutes). Others are related to the use of standards (Internet Inter-Object Request Broker Protocol of the Object Management Group) or underlying technology (Message Oriented Middleware), which are omitted just to make the specification readable. Finally, there are domain-specific properties that are deemed not necessary for inclusion, as they are well known by all developers. For example, in a transactional information system, transactions must be fully committed or fully rolled back; a state in which part of the transaction is committed and part rolled back is not acceptable.

As another example, consider the functional specification for caching information among processes [Clarke 95]. The requirements specification may state that when a process accesses the same cache entry as other processes, it will always indicate to the other processes when it modifies its local copy of that entry. The specification may also include a statement that all of the other processes sharing that line must invalidate their copies when another process has signaled its modification. In addition, other statements about invalidating, and so forth, may be included in the requirements specification. For a complex distributed system, these requirements may exist in very different parts of a large document or in multiple documents. Collectively, these requirements imply that "there is no case where a process thinks it has an unmodified copy when one or more other processes have modified the cache line." But a statement reflecting this condition (which can be considered an implied consequence) may not appear anywhere in the documentation. It can be useful to rely on personnel experiences and engineering expertise to identify properties that may not be explicitly stated in the requirements or other specification, but are important properties to maintain.

Another dimension to this issue is that the expected-property-generation process can uncover requirements that are not stated appropriately, are implied, or are sometimes omitted, but nevertheless should be included in a requirements document. For example, statements about

and responses to exception conditions during operation are often implied rather than explicitly stated in a requirements specification. In a requirements specification, there may be a statement that a data update process shall read the integer value between 0 and 10, inclusive, and modify it in some way. The value is sent by another process, but nowhere is it stated what to do if the value is outside the range or even not an integer. Similarly, in the mutual-exclusion example described earlier, the requirements document might explicitly state that: “no more than one process should access a resource at one time.” But a statement of fairness (e.g., that both processes must be able to gain access to resources when requested) might be omitted. This omission as well as the omission of the exception cases in the previous example should be viewed as deficiencies in a requirements specification<sup>2</sup>.

## 2.2 Application Domain Considerations

A particularly rich source of expected properties is application domain knowledge. These properties are founded upon general characteristics of systems that are common across the operational environment for the broad application domain of the system.

An interesting application domain for MBV techniques is the concurrent system. Concurrent systems have multiple processes running in parallel or interleaved. Each process has a particular behavior that may or may not conflict with the expected properties of the system as a whole. The following is a list of typical (and often implicit) global expected properties of concurrent systems.

- Fairness - All expected functions can execute or all processes have a chance to execute.
- Absence of deadlock - Deadlock is the condition where no progress is being made; for example, where processes are waiting for resources that are held by other non-executing processes and as a result none of the processes can execute.
- Absence of livelock - Livelock involves the condition where execution is occurring but no progress is seen; for example, a system does something but is not interacting with the environment.
- Absence of starvation - Starvation is the condition where one process dominates, not allowing any others to progress. The others are starved out.

## 2.3 Generic vs. Detailed Statements

As a software project evolves, the development team generates increasingly concrete specifications through successive refinements (or “realizations” in Object Management Group [OMG] terminology) at different levels of abstraction [OMG 01]. The user’s needs are refined into requirements; the requirements are refined into designs and so on. Model-based verification is used at different project phases to verify different artifacts. In each

---

<sup>2</sup> In current practice these implied requirements are often not included in a requirements document.

development phase, in addition to other sources such as requirements, users, customers, and so forth, expected properties should be generated from the source on which the modeled artifact is based. The design is used as the source of expected properties when verifying the code; the requirement specification is used to verify the design; and input from the users is used to verify the requirements.

In addition to the antecedent documents, general statements within a single specification can be used for generating expected properties. A general statement on which one or more detailed “refined” statements are based is not necessarily found in a predecessor document. Often, there are multiple levels of abstraction coexisting in the same document. For example, the very generic statement “no unauthorized user will be granted access to the system” found in the introduction of a requirements specification can be expanded into one or more sections of that specification with detailed descriptions of authentication protocols and standards. Typically, the detailed statements can be used as a source for models and the generic statements as a source for expected properties.

## 2.4 Whole vs. Part Statements

Modeling an entire system in detail is extremely difficult and generally impractical. For this reason, systems are normally decomposed into parts and each part is modeled separately. In modeling for a requirements specification, for example, different models are built for different parts of the system; each individual model represents the requirements belonging to that particular piece of the puzzle. Thereafter, we can argue that models are representing the individual parts and not the whole of the subject artifact.

If individual attributes (from parts) are the source for the models, we can use global attributes (for the whole) as well as implied characteristics of the individual parts as sources for expected properties. This approach verifies that the individual parts contribute to reach the global objectives. Returning to the security requirements specification example, we could verify that the individual security requirements of the different subsystems comply with the global security requirements for the system.

This strategy is particularly useful for Component-Based Software Engineering (CBSE) [Wallnau 00]. Component-based systems are ensembles of potentially numerous individual components. As the number of components increases, so do the number of possible interactions, which makes it very difficult to infer the properties of the ensemble from the properties of its components. MBV can be used to model how each individual component contributes to the behavior of the ensemble, thereby helping to verify that the expected properties of the ensemble hold.

---

## **3 Generating Expected Properties**

The generation of expected properties is part of an overall verification and validation strategy for the product. Expected properties focus model-based analyses on the critical system considerations: those aspects that are of special interest to the client, user, or other stakeholders, or those aspects of the system that are risky (e.g., safety or special reliability or other quality requirements).

### **3.1 Focusing the Generation Process**

In the context of Model-Based Verification, the identification of expected properties should be coordinated with the generation of the statement of scope, formalism, and perspective [Gluch 01].

- **Scope:** the portion of a system that is to be modeled and analyzed is its scope. The critical (important and/or risky) aspects of the system and its development, including both programmatic and technical issues, are used to define the scope.
- **Formalism:** the modeling approaches and tools to be used. Modeling techniques that can be employed in Model-Based Verification include state machine representations, process algebras, and rate monotonic modeling.
- **Perspective:** the context for representing the behavior and characteristics of a system. A perspective could be the user's view of a system or it could be the representation of a specific feature, function, or capability of a system.

The perspective and scope focus the analysis efforts on the important or risky elements of the system. Expected properties specifically identify the behaviors associated with those aspects and parts of the system highlighted in the perspective and scope.

Scope and perspective can be used to make the expected-properties-generation process iterative. Initially, expected properties for a small component (scope) related to a particular perspective can be created. Later, more components can be added or more perspectives can be considered. This incremental and iterative approach can be followed until a satisfactory coverage of the system is achieved.

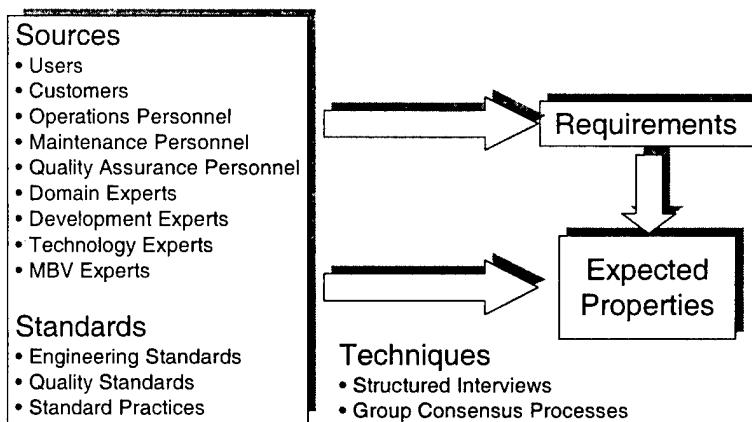
### **3.2 Partitioning of Expected Properties**

To further structure expected properties and their elicitation and capture process, the system can first be examined with respect to 1) what is expected or desired of its behavior, and 2) what is not desired in its behavior.

1. Desired (what should happen) - Examine the behavior scenarios of the system and identify how it should be working and what must happen for the system to work properly, capturing specific statements of this type of behavior.
2. Undesired (what should not happen) - Examine the behavior scenarios of the system and explicitly identify what things should not happen and what could happen to result in undesirable system behavior.

### 3.3 Capturing Expected Properties

The process of capturing expected properties resembles that of capturing requirements [Siddiqui 96]. Part of the elicitation flow for generating expected properties is shown in Figure 3. Expected properties can be collected directly from stakeholders employing similar techniques to requirements elicitation. Alternatively, expected properties can be extracted directly from requirements documents and related specifications.



*Figure 3: Requirements and Expected Properties Elicitation Processes*

In addition to a similar pattern, both processes involve multiple stakeholders, can be facilitated by team processes (consensus-based practices), and are directed towards getting the key aspects of the system identified. As with requirements elicitation and capture, it is important with expected properties to ensure that a variety of views and stakeholders are represented in the process, regardless of the specific method used. The following list enumerates different sources for expected properties.

1. *Requirements documents:* If the project has followed sound engineering practices, requirements should be the most reliable source of expected properties, as they are the consensus of what the system is supposed to do. One important value of generating expected properties from requirements specifications is that it can uncover requirements that are not stated appropriately, are implied, or are omitted.

2. *Users of the system, customers, and operations and maintenance personnel:* An excellent source of information about what the system is supposed to do is the collective knowledge of stakeholders including users, customers, and operations and maintenance personnel. Members of these groups can provide first-hand insight into behavioral aspects of systems. If sound engineering practices were used in the original development effort, all of these groups would have had a role in defining the requirements documentation.
3. *Domain experts and quality assurance personnel:* We have grouped in this category sources that are not specific to the system. There are expected properties common to individual application domains. For example, every entry in the assets column of a balance sheet should be followed by an entry in the liabilities column. Other expected properties represent quality attributes common to any software system. For example, all internal errors should produce some defined and identifiable external manifestation.
4. *Developers and implementation technology experts:* In theory, technical details of the implementation are not needed to define the expected properties of the system. In practice, however, an understanding of the internals of a potential or actual implementation of the system can be useful in generating effective expected properties.
5. *MBV experts:* The activity of translating specifications into models can result in errors. Even if these errors do not affect the quality of the system, they do hamper the capability of models to faithfully reflect the behavior of the system. There are expected properties that address the correctness of the models in contrast to the correctness of the system. For example, every state in a state machine should be reachable under some sequence of valid input. If a state is not reachable it is cluttering the model without reflecting any feasible system behavior. MBV experts are software engineers who are trained and experienced in applying Model-Based Verification techniques in a variety of application domains. They can help to craft expected properties that are more readily expressed as formal statements, thereby facilitating and reducing errors in the translation of expected properties into claims.
6. *Standards:* There are expected properties induced by the standards with which the system must comply. For example, if the system uses Common Object Request Brokers Architecture (CORBA) Messaging v. 2.4.2 with order policy set up to ORDER\_TEMPORAL, the messages from the client should be guaranteed to be processed in the same order in which they were sent.

---

## 4 Expected Properties and Claims

To be used in automated model checking, the natural language statements of expected properties must be expressed as formal statements—claims. The various translations among requirements and claims are shown in Figure 4. There are three basic paths to formal claims.

In most cases natural language statements of expected properties, based primarily upon domain expertise, are generated first. Software engineers who are experts in the relevant formal language translate these statements into formal expressions for model checking<sup>3</sup>. This translation is often not a one-to-one mapping of statements. The richness and ambiguities of natural language, in contrast to the precision and constraints of a formal style, often make the translation difficult and error prone. Consequently, the translation process should also involve domain experts and others who helped to develop the expected property statements. Their involvement can be either in direct support of the process or as active reviewers. In an active reviewer role, these individuals interact with software engineers to establish a shared interpretation of the formal statements, one where all parties agree on the consistency of the intent between the natural language statements and their formal expression(s). This collective involvement will help to ensure that subtle differences between the languages do not result in formal statements that misrepresent the expected property statement.

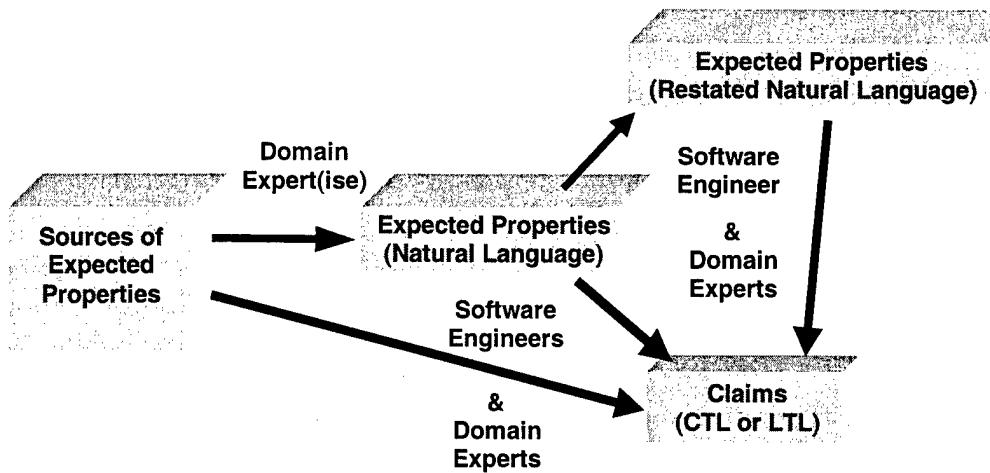
Occasionally, it can be valuable to restate the expected property in a form that is more amenable to direct translation. This approach can be used as a method to clarify the meaning for domain experts who are not conversant with the formal language. This may be a semiformal structured expression. An intermediate representation can facilitate a review process. It can also enhance the understanding of the domain issues by software engineers involved in the process, and of the formal expressions by other domain participants in the process. For example, consider an expected property: “All alarms will be displayed at the system console.” This may be broken into an intermediate statement: “If an alarm condition is detected then it must be displayed.” This is readily translated into the Computation Tree Logic (CTL) expression  $AG(alarm \rightarrow AF\ display)$ . In addition it exposes the need to confirm that, in the model, the alarm can eventually occur (i.e., EF alarm).

The most direct path is where formal claims are developed from the sources used to generate expected properties (e.g., a system description or requirements document). In this case software engineers, expert in the relevant formal language (e.g., CTL or Linear Temporal Logic [LTL]), formulate the claims, often in cooperation with domain experts. This approach

---

<sup>3</sup> For more information on the issues associated with the translation of expected properties into formal representations, see the technical note by Comella-Dorda and associates [Comella-Dorda 01].

can work if the software engineer generating the claims is also an expert in the domain. As is the case with the other paths, it is important that other individuals be involved, either directly or as reviewers of the natural language statements and their formal representations, to help to ensure the correctness of the claims.



*Figure 4: Expected Property Transformations*

This technical note addresses the creation of the natural language statements (expected properties). Since expected properties are natural language expressions, they can have arbitrary structures and semantics. The transformation of expected properties into claims is not covered in this report. However, we do include an overview of the relation between expected properties and claims since understanding this relation is important in the effective analysis of formal models. For additional information on this relationship and the generation of formal claims consult the technical note: *Model-Based Verification: Claim Creation Guidelines* [Comella-Dorda 01].

#### 4.1 The Relationships Between Expected Properties and Claims

General statements of what is expected of the system are often not readily translatable into meaningful claims. Any kind of expression that defines what is good (or bad) for a system can be considered an expected property. For example, the expression “the system works properly under any condition” is a valid expected property; unfortunately it cannot be immediately formalized into a realistic claim to be verified by a model-checking tool. Only by defining specifically, in the context of the model, what the correct operations are under each specific condition can a useable expected property and resultant claim be produced.

Other variations that can be realized between expected property statements and claims involve the practice of iteratively exploring expected properties by using the flexibility in the form of the claims. For example, in analyzing a model it is often useful to start with a claim

that is weaker than the expected property and create additional stronger claims until reaching the semantic level of representation of the expected property. For example, suppose that an expected property is “The value of variable E must always decrease.” In checking a model we can start with the claim:

$EF(E.\text{decreasing})$  = The value of variable E must be able to decrease at least once

After verifying the weak claim, we can strengthen it to read

$AGAF(E.\text{decreasing})$  = The value of variable E must be able to decrease multiple times (an infinite number of times)

If this is confirmed as true, we will reach the semantic level of the original expected property

$AG(E.\text{decreasing})$  = The value of variable E must always decrease

## 4.2 Classifying Expected Properties and Claims

When trying to classify expected properties, it is useful to focus on the range in which the property (and the claim) applies. According to this classification, we identify the following types of expected properties.

- invariants (global and local) - conditions that remain unchanged throughout (globally) or conditions that remain unchanged in a particular portion or under specific circumstances (locally)
- assertions - conditions that apply at a single point in an execution<sup>4</sup>
- pre- and post-conditions - condition pairs that apply before and after something occurs in a system. These can be viewed as assertion pairs.
- constraints - statements that apply across all changes or a subset of changes that occur in the system

This classification and the process of generating expected properties are influenced, in part, by the realization that the natural language statements must be translated into formal representations—claims. Consequently, while the generation process should rely on the guidelines outlined earlier, using the four classifications defined above during the identification process can help to tailor the expression of expected properties into forms that more readily translate into formal expressions. Invariants, assertions, pre-and post-conditions, and constraints are propositions, in that they can be evaluated as either true or false when applied to a model. Thinking about the systems’ behavioral characteristics in the context of these propositional forms can help in phrasing natural language statements that are more amenable to direct translation into a formal representation. For example, in a traffic light

---

<sup>4</sup> A point invariant is equivalent to an assertion and is a single-line local invariant. A local invariant may span more than one state or point of execution, e.g. a loop invariant in a program. Invariants talk about things that don’t change and assertions are about what is true at a single point in an execution.

control system that encompasses a large metropolitan area, it is clear that the system should control all intersections so that only one traffic flow direction is permitted through that intersection at a time. In thinking about what should not happen with respect to this property, one can identify an invariant stating that in any intersection in the system: “It will never be the case that the north-south traffic light and the east-west traffic light are both simultaneously green.” This is readily expressed as a formal claim in CTL: AG !(N\_S = green & E-W = green).<sup>5</sup> Employing a usage-based approach complemented by considerations of the ultimate need to formalize the expected properties can facilitate the model-checking process.

## Invariants

Global invariants remain unchanged throughout all possible behaviors of the system. These forms can be used to express characteristics of the system that remain constant throughout all executions [Bensalem 96]. Often these are not explicitly stated in a requirements specification. For example, in a flight control system, the forward loop gain may take on many values. These may depend on the airspeed or some other factor, but whatever the value, it is never negative or equal to zero.. Thus, an invariant of the system is that the “Forward control loop\_gain is always greater than zero.” Another example, in a complex system consisting of multiple operating modes, might be that it is always possible for the system to return to the idle mode.

As another example, consider a process control system and the statement that while the system is in mixing mode, the secondary and primary flow control systems are operational. The entire statement is true throughout all executions. It is a global invariant, but it can be viewed as consisting of a local invariant with its restricting condition. Local invariants are statements that apply only at certain times, over a range of states or executions of the system. In the example, the local invariant is the statement “the secondary and primary flow control systems are operational” which is restricted by the condition that “the system is in mixing mode.”

One approach for identifying invariants is to look for distinct conditions or collective (related) conditions among components that must be true no matter what happens. This may involve looking at system operational scenarios (e.g., detailed use cases in O-O representations) and questioning what characteristics should remain unchanged. This questioning is embedded in the two-fold usage-based procedures of iteratively considering what is desired and what is undesired in the behavior.

---

<sup>5</sup> Comella-Dorda and associates provide more details on the CTL notation and claims [Comella-Dorda 01].

## **Assertions**

Assertions are statements that apply at some specific point (instant) in the execution. For example, when the airspeed is 100 knots and the altitude is 1000 feet, the forward loop gain variable is 0.8. Assertions are widely used in programming languages to check the values of variables during the execution [Drabent 98, Rosenblum 95]. As an example consider the simple assertion “assert( speed  $\geq 0$ )” inserted in a flight navigation program. It checks that the variable “speed” is non-negative at that point of execution.

## **Pre- and Post-Conditions**

Pre- and post-conditions are the best-known kind of assertions. They are used to assess the impact of some computational or execution element in a system. Their semantics are the following: “If the pre-condition is true before the execution, then the post-condition must be true after the execution. If the pre-condition is not true, the result is undefined.” As a simple example, consider driving. A pre-condition of leaving a parking space would be that the engine is running. A post-condition for leaving a parking space would be that the car is not parked anymore.

## **Constraints**

Constraints are similar to invariants but they operate over the transitions of the system rather than over the states. They describe restrictions on changes that may occur in the system. For example, a constraint for a telecommunications system is that when the number of clients increases, the number of active server connections must increase. As with invariants, constraints can be applied globally or locally over executions of the system.

One approach for identifying constraints is to look for limits on the variations that are possible for critical system parameters throughout the system’s execution. As an example, consider a chemical-process-control-system specification. The specification requires that while the system is in the mixing mode, a key feedback parameter affecting process concentrations must never decrease. A constraint over all mixing-mode transitions would be that the feedback parameter must always remain constant or increase.

---

## 5 Conclusions

Defining the characteristics of behavior in natural language is a difficult task. Often, no single individual has a clear idea of all of the expected properties of the system being developed. While requirements specifications are one of the most prominent sources of expected properties, important system properties can be poorly defined in, or completely missing from, a requirements specification. Consequently, effective expected-property-generation processes require

- collaboration among all of the stakeholders involved
- recognition of various levels of refinement and scope throughout a development effort
- considerations of what might not be explicitly documented
- what is generally implied by the application environment or a specific user need

The inherent challenge associated with identifying expected properties is further complicated by the need to eventually represent them as formal statements for an automated model-checking tool. Consideration of this need by thinking about formal expression categories and forms within the generation process can facilitate the entire model-checking activity. These considerations will enable easier translation and can help to guide the identification process (e.g., looking for what is true always).

Throughout all of the activities involved in defining expected properties, the main assets for the practitioner are a thorough understanding of both the system and the domain, and a reliance on sound engineering principles that guide the development of any successful software system.

Taking the time to define expected properties can provide deeper insight into a system and its design, and identify potential defects, even if no formal model checking is performed. This technical note presented a number of guidelines and techniques to facilitate the critical activity of expected property definition. As Model-Based Verification practices mature, modifications and extensions to the guidelines introduced in this technical note will be developed.



---

## References

- [Bensalem 96] Bensalem, S.; Lakhnech, Y.; & Saidi, H. "Powerful Techniques for the Automatic Generation of Invariants." *Computer Aided Verification, 8th International Conference, CAV '96*, New Brunswick, NJ, July 31 - August 3. *Lecture Notes in Computer Science, 1102*. New York, NY: Springer-Verlag, 1996.
- [Clarke 95] Clarke, E. M.; Grumberg, O.; Hiraishi, H.; Jha, S.; Long, D.; McMillan, K. L.; & Ness, L. A.. "Verification of the Futurebus+ Cache Coherence Protocol." *Formal Methods in System Design 6*, (1995): 217-232.
- [Clarke 96] Clarke, E. M. & Wing, J. "Formal Methods: State of the Art and Future Directions." *ACM Computing Surveys 28, 4* (December 1996): 626-643.
- [Comella-Dorda 01] Comella-Dorda, S.; Gluch, D. P.; Hudak, J; Lewis, G; & Weinstock, C. *Model-Based Verification: Claim Generation Guidelines* (CMU/SEI-2001-TN-018). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001.  
<<http://www.sei.cmu.edu/publications/documents/01.reports/01tn018.html>>.
- [Drabent 98] Drabent, W. ; Nadjm-Tehrani, S.; & Maluszynski, J. "Algorithmic Debugging with Assertions." Abramson H. & Rogers, M. H, ed. *Meta-programming in Logic Programming*. Cambridge, MA: MIT Press, 1989.
- [Gluch 98] Gluch, D. P. & Weinstock, C.B. *Model-Based Verification: A Technology for Dependable Upgrade* (CMU/SEI-98-TR-009, ADA354756). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1998. <<http://www.sei.cmu.edu/publications/documents/98.reports/98tr009/98tr009abstract.html>>.
- [Gluch 01] Gluch, D. P.; Comella-Dorda, S.; Hudak, J; Lewis, G; & Weinstock, C. *Model-Based Verification: Scope, Formalism, and Perspective Guidelines*. Pittsburgh, PA: Software Engineering Institute, Carnegie

- Mellon University. 2001. <<http://www.sei.cmu.edu/publications/documents/01.reports/01tn024.html>>.
- [Jackson 96a]** Jackson, D. “Nitpick: A Checkable Specification Language.” *Proceedings of the Workshop on Formal Methods in Software Practice*. San Diego, CA, January 10-11, 1996. New York, NY: ACM Press, 1996.
- [Jackson 96b]** Jackson, D. & Damon, C. *Nitpick Reference Manual*. Pittsburgh, PA: Carnegie Mellon University, 1996. <<http://www2.cs.cmu.edu/afs/cs/project/nitpick/www/home.html>>.
- [Jackson 00]** Jackson, D. “Alloy: A Lightweight Object Modelling Notation.” Cambridge, MA: Massachusetts Institute of Technology, 2000. <<http://sdg.lcs.mit.edu/~dnj/publications.html>>.
- [McMillan 92]** McMillan, K.L. *Symbolic Model Checking: An Approach to the State Explosion Problem* (CMU-CS-92-131). Pittsburgh, PA: Computer Science Department, Carnegie Mellon University, 1992.
- [OMG 01]** Object Management Group. *Unified Modeling Language Specification* (draft). <<ftp://ftp.omg.org/pub/docs/ad/01-02-14.pdf>> Version 1.4 draft (February 2001).
- [Rosenblum 95]** Rosenblum, D.S. “A Practical Approach to Programming with Assertions.” *IEEE Transactions on Software Engineering* 21, 1 (January 1995): 19-31.
- [Siddiqui 96]** Siddiqui, J. & Shekaran M. C. “Requirements Engineering: The Emerging Wisdom.” *IEEE Software* 13, 2 (March 1996): 15-19.
- [Wallnau 00]** Bachmann, F.; Bass, L.; Buhman, C.; Comella-Dorda, S.; Long, F.; Robert, J.; Seacord, R.; & Wallnau, K. *Volume II: Technical Concepts of Component-Based Software Engineering* (CMU/SEI-2000-TR-008 ADA379930) Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <<http://www.sei.cmu.edu/publications/documents/00.reports/00tr008.html>>.

## REPORT DOCUMENTATION PAGE

*Form Approved  
OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY  (Leave Blank)	2. REPORT DATE  January 2002	3. REPORT TYPE AND DATES COVERED  Final	
4. TITLE AND SUBTITLE  Model-Based Verification: Guidelines for Generating Expected Properties		5. FUNDING NUMBERS  F19628-00-C-0003	
6. AUTHOR(S)  David P. Gluch, Santiago Comella- Dorda, John Hudak, Grace Lewis, Chuck Weinstock			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)  Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER  CMU/SEI-2002-TN-003	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)  HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES			
12A DISTRIBUTION/AVAILABILITY STATEMENT  Unclassified/Unlimited, DTIC, NTIS		12B DISTRIBUTION CODE	
13. ABSTRACT (MAXIMUM 200 WORDS)  This report presents a basic set of guidelines to facilitate the generation of expected properties in the context of Model-Based Verification. Expected properties are natural language statements that express characteristics of the behavior of a system—characteristics that are consistent with user expectations. Through model checking, expected properties of a system, formally expressed as claims, are analyzed against the model. This analysis can detect inconsistencies between models of the system and their expected properties and identify potential system defects.			
14. SUBJECT TERMS  Model-Based Verification, expected properties, claims		15. NUMBER OF PAGES  26	
16. PRICE CODE			
17. SECURITY CLASSIFICATION OF REPORT  Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE  Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT  Unclassified	20. LIMITATION OF ABSTRACT  UL